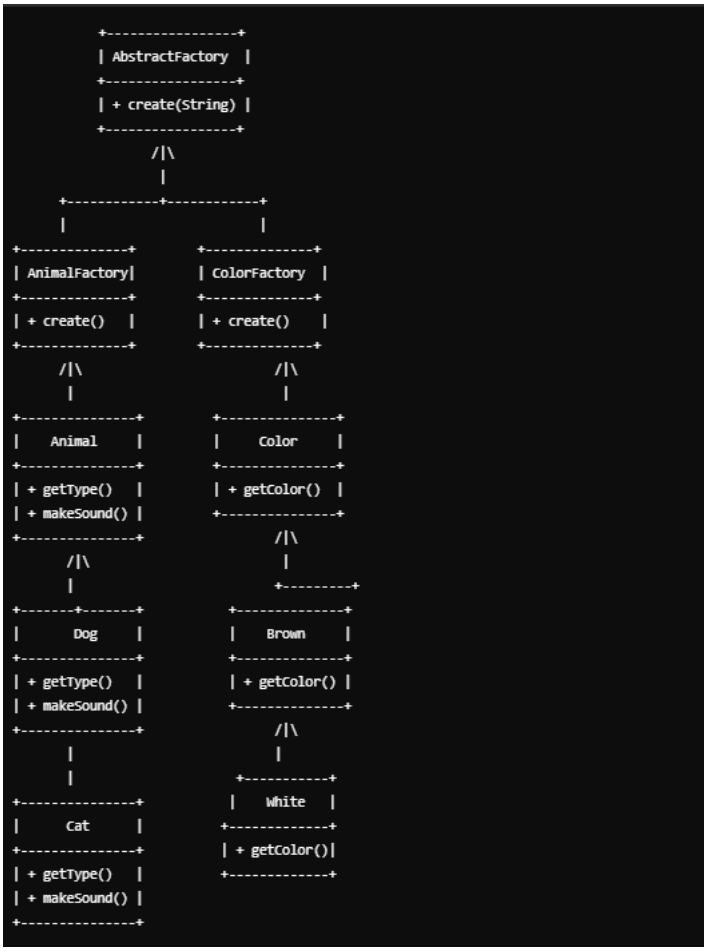
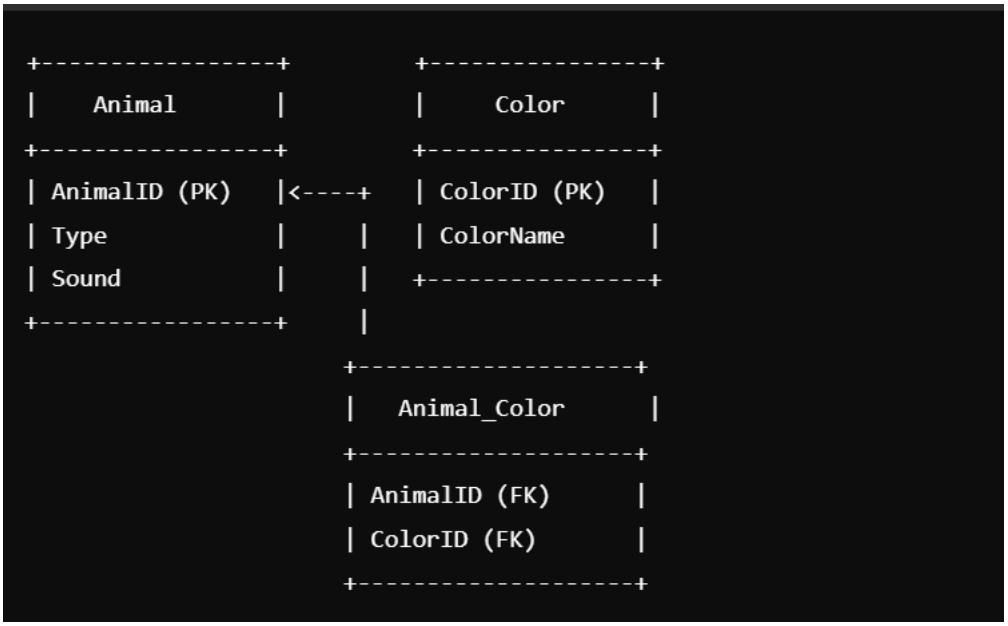
1. Class Diagram:



2. . Entity-Relationship (ER) Diagram



Abstract Factory Layer Diagram:

Explanation

- **Class Diagram**: This illustrates the relationships between interfaces, concrete classes, and the methods they implement. The AbstractFactory interface is implemented by AnimalFactory and ColorFactory, which create Animal and Color objects respectively.

- **ER Diagram**: This shows the relationship between entities in a database context. The Animal and Color entities are linked through a join table (Animal_Color), which connects the primary keys of both entities.

- **Abstract Factory Layer Diagram**: This represents the layers of the abstract factory pattern, showing how the client interacts with the factory and the various product families.

Code Explanation:

1. Abstract Factory Interface

java
Copy code
```
public interface AbstractFactory<T> {
    T create(String type);
}
```

- This interface defines a method create that takes a string (type) and returns an object of type T. This allows for the creation of various product families.

2. Product Interfaces

- **Animal Interface**

java
Copy code
```
public interface Animal {
    String getType();
    String makeSound();
}
```

- This interface defines the behavior expected from any animal, such as getting its type and making a sound.
- **Color Interface**

java
Copy code
```
public interface Color {
    String getColor();
}
```

- This interface defines the method for retrieving the color.

3. Concrete Classes for Animals

- **Dog Class**

java
Copy code
```
public class Dog implements Animal {
    @Override
    public String getType() {
        return "Dog";
    }

    @Override
    public String makeSound() {
        return "Bark";
    }
}
```

- This class implements the Animal interface, providing the type and sound of a dog.
- **Cat Class**

java
Copy code
```
public class Cat implements Animal {
    @Override
```

```
    public String getType() {
        return "Cat";
    }

    @Override
    public String makeSound() {
        return "Meow";
    }
}
```

- Similar to Dog, this class provides the type and sound for a cat.

4. Concrete Classes for Colors

- **Brown Class**

```
java
Copy code
public class Brown implements Color {
    @Override
    public String getColor() {
        return "Brown";
    }
}
```

- Implements the Color interface to provide the color "Brown".
- **White Class**

```
java
Copy code
public class White implements Color {
    @Override
    public String getColor() {
        return "White";
    }
}
```

- Implements the Color interface to provide the color "White".

5. Concrete Factories

- **AnimalFactory**

```
java
Copy code
public class AnimalFactory implements AbstractFactory<Animal> {
    @Override
    public Animal create(String animalType) {
        if ("Dog".equalsIgnoreCase(animalType)) {
            return new Dog();
        } else if ("Cat".equalsIgnoreCase(animalType)) {
            return new Cat();
        }
        return null;
    }
}
```

- This factory creates Animal objects based on the input type. It checks the type and returns an instance of either Dog or Cat.

- **ColorFactory**

  java
  Copy code
  ```
  public class ColorFactory implements AbstractFactory<Color> {
     @Override
     public Color create(String colorType) {
        if ("Brown".equalsIgnoreCase(colorType)) {
           return new Brown();
        } else if ("White".equalsIgnoreCase(colorType)) {
           return new White();
        }
        return null;
     }
  }
  ```
- Similar to AnimalFactory, it creates Color objects based on the specified type.

  6. Factory Provider

  java
  Copy code
  ```
  public class FactoryProvider {
     public static AbstractFactory<?> getFactory(String choice) {
        if ("Animal".equalsIgnoreCase(choice)) {
           return new AnimalFactory();
        } else if ("Color".equalsIgnoreCase(choice)) {
           return new ColorFactory();
        }
        return null;
     }
  }
  ```
- This class acts as a factory provider, returning the appropriate factory (either AnimalFactory or ColorFactory) based on user choice.

  7. Client Code (Usage)

  java
  Copy code
  ```
  public class AbstractFactoryPatternDemo {
     public static void main(String[] args) {
        // Get Animal Factory
        AbstractFactory<Animal> animalFactory = (AbstractFactory<Animal>)
  FactoryProvider.getFactory("Animal");
        Animal animal1 = animalFactory.create("Dog");
        System.out.println("Animal: " + animal1.getType() + ", Sound: " + animal1.makeSound());

        // Get Color Factory
        AbstractFactory<Color> colorFactory = (AbstractFactory<Color>)
  FactoryProvider.getFactory("Color");
        Color color1 = colorFactory.create("Brown");
        System.out.println("Color: " + color1.getColor());
     }
  }
  ```

- This is the main method demonstrating how to use the abstract factory pattern. It first retrieves the AnimalFactory, creates a Dog, and prints its type and sound. Then, it retrieves the ColorFactory, creates a Brown color, and prints its color.
  Output
  yaml
  Copy code
  Animal: Dog, Sound: Bark
  Color: Brown
  Summary
  The Abstract Factory design pattern allows you to create families of related objects without specifying their concrete classes. This promotes flexibility and maintainability:
- **Encapsulation**: Clients interact with factories instead of concrete classes, decoupling the client code from the specifics of the object creation.
- **Ease of Extension**: New animals or colors can be added by simply creating new classes and updating the factories without changing existing code.
- **Consistency**: Ensures that products from a family (e.g., animals and colors) are used together, maintaining logical coherence in the application.